Analysing Snapshot Isolation

Andrea Cerone
IMDEA Software Institute

Alexey Gotsman
IMDEA Software Institute

ABSTRACT

Snapshot isolation (SI) is a widely used consistency model for transaction processing, implemented by most major databases and some of transactional memory systems. Unfortunately, its classical definition is given in a low-level operational way, by an idealised concurrency-control algorithm, and this complicates reasoning about the behaviour of applications running under SI. We give an alternative specification to SI that characterises it in terms of transactional dependency graphs of Adya et al., generalising serialization graphs. Unlike previous work, our characterisation does not require adding additional information to dependency graphs about start and commit points of transactions. We then exploit our specification to obtain two kinds of static analyses. The first one checks when a set of transactions running under SI can be chopped into smaller pieces without introducing new behaviours, to improve performance. The other analysis checks whether a set of transactions running under a weakening of SI behaves the same as when it running under SI.

Keywords

Snapshot isolation; transaction chopping; robustness

1. INTRODUCTION

Transactions simplify concurrent programming by enabling computations on shared data that are isolated from other concurrent computations and resilient to failures. They are commonly provided by databases [7] and, more recently, by transactional memory systems [22]. Ideally, programmers would like to get strong guarantees about the isolation of transactional computations, formalised by the notion of serializability [7]: the results of concurrently executing a set transactions could be obtained if these transactions executed atomically in some order. Unfortunately, ensuring serializability carries a significant performance penalty. For this reason, transactional systems often provide weaker guarantees about transaction processing, formalised by weak consistency models. Snapshot isolation (SI) [6] is one of the most popular such models, implemented by major centralised databases (e.g., MS SQL Server, Oracle), distributed databases [14, 27, 29] and transactional memory systems [1, 8, 15, 25].

Informally, SI is defined by a multi-version concurrency control algorithm as follows. A transaction T reads values of shared objects from a snapshot taken at its start. The transaction commits only if it passes a write-conflict detection check: since T started, no other committed transaction has written to any object that T also wrote to. If the check fails, T aborts. Once T commits, its changes become visible to all transactions that take a snapshot afterwards. This concurrency-control algorithm allows unserializable behaviours, called anomalies. One of them, write skew, is graphically illustrated in Figure 2(d). Each of the transactions T1 and T2 checks that the combined balance of two accounts exceeds 100 and, if so, withdraws 100 from one of them. Under SI, both transactions may pass the checks and make the withdrawals from different accounts, resulting in the combined balance going negative. This outcome cannot occur under serializability. Given such anomalies, reasoning about the behaviour of applications executing under SI is far from trivial. This task is further complicated by the fact that the specification of SI is given in a low-level operational way, by a concurrency control algorithm. To facilitate reasoning about applications using SI and establishing useful results about this consistency model, we need a more declarative specification that abstracts from implementation-level details as much as possible.

An approach that yields such consistency model specifications was proposed by Adya et al. [2, 3]. In this approach, an execution of a set of transactions is described by three kinds of dependencies between pairs of transactions T1 and T2: read dependencies record when T1 reads the value of an object written by T2; write dependencies record when T1 overwrites the value of an object written by T2; finally, anti-dependencies are derived from read and write dependencies in a certain way (§3). A set of transactions and dependencies between them form a dependency graph, generalising classical serialization graphs [7]. Then the set of executions allowed by a given consistency model is defined by those dependency graphs that lack certain cycles; in particular, serializable executions are characterised by acyclic dependency graphs. This way of specifying consistency models has been shown to be particularly appropriate for designing static analyses [12, 19, 23, 30, 38], run-time monitoring [9, 37] and proving concurrency-control algorithms correct [16, 24, 36]. In particular, specifications in terms of dependency graphs facilitate exploring possible program executions in a static analysis, because the analysis can determine which dependencies can possibly exist at run time by looking for pairs of read or write accesses to the same object in the code of different transactions. In contrast, it is hard to predict statically more low-level information about transaction execution, such as the order in which transactions commit.

Specifications in terms of dependency graphs have been proposed for ANSI isolation levels such as serializability, Read Committed and Repeatable Read [2], as well as more recent proposals of consistency models [5, 36]. But surprisingly, there is no such spec-
ification of SI. This is not for the want of trying: Adya did propose a definition of SI that refers to dependency graphs [2]. However, to capture the subtle semantics of SI, this definition extends the graphs by a relation describing low-level information about transaction execution, which negates their benefits.

In this paper we propose the first characterisation of SI solely in terms of dependency graphs (§4) and apply it to develop new static analyses (§5 and §6). Namely, we show that SI allows exactly the executions represented by dependency graphs that contain only cycles with at least two adjacent anti-dependency edges. The proof of this fact is highly non-trivial and represents a key technical contribution of this paper. It requires showing that, given a dependency graph satisfying the above acyclicity condition, we can construct certain relations describing how the transactions can be processed by the SI concurrency control, e.g., the order in which transactions commit. Constructing these relations from transactional dependencies is challenging, and the main insight of our proof is given by a procedure for this construction, based on solving certain kinds of inequalities over relations.

To illustrate the benefits of our dependency graph characterisation of SI, we exploit it to develop two kinds of static analyses. First, we propose a new static analysis for the classical problem of transaction chopping [4, 30, 35]—checking when transactions in an application can be chopped into smaller pieces without introducing new behaviours (§5). When applied to long-running transactions executing under SI, chopping can improve performance, because the longer an SI transaction runs, the higher the chances are that it will abort due to a write conflict. There are analyses for transaction chopping under serializability [30] and parallel SI [12], a recently-proposed weaker version of SI for large-scale systems that do not change significantly during splicing, and this makes it easy to construct the spliced execution.

The other kind of static analyses that we consider checks whether an application is robust [19, 31] against weakening consistency: it behaves the same regardless of whether it uses a database providing a weak consistency model or a database proving a stronger model (§6). When this is the case, the application programmer can reap the performance benefits of using the weaker model, yet can reason about the correctness of the application assuming the stronger one. We first show that our SI characterisation allows easily deriving a variant of an existing analysis that checks whether an application executing under SI behaves the same as when executing under serializability [19] (robustness against SI, §6.1). We then propose a new static analysis that checks whether an application executing under the recently-proposed parallel SI [32] behaves the same as when executing under the stronger classical SI (robustness against parallel SI towards SI, §6.2). To derive this static analysis, we formulate a dependency graph characterisation of parallel SI, which can be given more easily than for classical SI. Again, our characterisations of consistency models in terms of dependency graphs greatly facilitate deriving the above robustness analyses, since the characterisations allow us to easily map between executions on different models.

Due to space constraints, we defer some of the proofs to [11, §A].

2. SNAPSHOT ISOLATION

We start by formally defining snapshot isolation (SI), as well as serializability. Rather than using the classical definition of SI by a concurrency-control algorithm (§1), it is technically convenient for us to build on a more declarative specification that we previously proposed and proved equivalent to the standard one [10]. Even though this specification is stated in terms of lower-level relations than transactional dependencies, it avoids referring explicitly to times at which a transaction takes a snapshot in the SI concurrency-control algorithm. We first introduce mathematical structures that represent transaction execution in the specification.

We consider a transactional system managing a set of integer-valued objects $\text{Obj} = \{x, y, \ldots\}$. Transactions read and write the objects, and in our representation of executions, we denote each invocation of such an operation by an event $e$. A function $\text{op} : \text{Event} \to \text{Op} = \{\text{read}(x, n), \text{write}(x, n) \mid x \in \text{Obj}, n \in \mathbb{Z}\}$ determines the operation a given event denotes: reading a value $n$ from an object $x$ or writing $n$ to $x$.

We call a binary relation a strict partial order if it is transitive and irreflexive. We call it a total order if it additionally relates any pair of distinct elements one way or another. We represent an execution of a single transaction by the following structure, recording a set of operations and the order in which they were invoked.

**Definition 1.** A transaction $T, S, \ldots$ is a pair $(E, \text{po})$, where $E \subseteq \text{Event}$ is a finite, non-empty set of events and the program order $\text{po} \subseteq E \times E$ is a total order.

For simplicity, all transactions in this paper are assumed to be committed: our specifications do not constrain values read inside aborted or ongoing transactions; this limitation could be lifted following [2, 17, 21]. We denote components of transactions and similar structures as in $E_T$ and $\text{po}_T$.

To allow transaction chopping (§5), we assume that the transactional system allows its clients to group several transactions into a session [33], which establishes an ordering on the transactions. Thus, instead of classical SI and serializability, we actually define their strong session variants [13, 14]. We represent the client-visible results of an execution of a set of sessions by a history.

**Definition 2.** A history is a pair $\mathcal{H} = (T, \text{SO})$, where $T$ is a finite set of transactions with disjoint sets of events and the session order $\text{SO} \subseteq T \times T$ is an union of total orders defined on disjoint subsets of $T$, which correspond to transactions in different sessions.

For simplicity, we elide the treatment of infinite computations, and thus histories are always finite. A consistency model, such as SI or serializability, is specified by a set of histories. To define this set, we extend histories with two relations, declaratively describing how the transactional system processes transactions.

**Definition 3.** An abstract execution (or just an execution) is a tuple $X = (T, \text{SO}, \text{VIS}, \text{CO})$, where $(T, \text{SO})$ is a history and the visibility and commit orders $\text{VIS}, \text{CO} \subseteq T \times T$ are such that $\text{VIS} \subseteq \text{CO}$ and $\text{CO}$ is total.
∀(E, po) ∈ T. ∀e ∈ E. ∀x, n. op(e) = read(x, n) \land \{ f \mid op(f) = \_ (x, n) \land f \not\triangleright e \} \neq \emptyset \implies op(maxpo\{ f \mid op(f) = \_ (x, n) \land f \not\triangleright e \}) = \_ (x, n) \quad (\text{INT})\]

\[
\forall T \in T. \forall x, n. T \vdash \text{read}(x, n) \implies \max_{\text{CO}}(\text{VIS}^{-1}(T) \cap \text{WriteTx}_a) \vdash \text{write}(x, n) \quad (\text{EXT})
\]

\[
\forall T, S \in T. \forall x. (T, S \in \text{WriteTx}_a \land T \neq S) \implies (T \not\triangleright \text{VIS} \land \forall S \not\triangleright \text{VIS} \rightarrow T) \quad (\text{NoConflict})
\]

\[\begin{align*}
(a) \text{ Session guarantees.} & \quad (d) \text{ Write skew. Initially acct} \_1 = \text{acct} \_2 = 60. \\
T_1 & \quad \text{ if (acct} \_1 + \text{acct} \_2 > 100) \\
& \quad \text{ acct} \_1 := \text{acct} \_1 - 100 \\
& \quad \text{ T}_1 \rightarrow \text{ read}(x, 1) \\
& \quad \text{ acct} \_2 := \text{acct} \_2 - 100 \\
& \quad \text{ T}_2 \rightarrow \text{ write}(x, 1) \\
& \quad \text{ acct} := \text{acct} + 50
\end{align*}\]

\[\begin{align*}
(b) \text{ Lost update.} & \quad (c) \text{ Long fork.} \\
\text{ T}_1 & \quad \text{ read}(\text{acct}, 0) \rightarrow \text{write}(\text{acct}, 50) \\
& \quad \text{ RW, WW} \\
& \quad \text{ VIS} \\
& \quad \text{ RW, WR} \\
\text{ T}_2 & \quad \text{ read}(\text{acct}, 0) \rightarrow \text{write}(\text{acct}, 25) \\
& \quad \text{ VIS} \\
& \quad \text{ WR} \\
\text{ T}_3 & \quad \text{ read}(\text{acct}, 0) \rightarrow \text{write}(\text{acct}, 25) \\
& \quad \text{ VIS} \\
& \quad \text{ WR} \\
& \quad \text{ VIS} \\
\end{align*}\]

\[\begin{align*}
& \quad \text{ (a) Session guarantees.} \\
& \quad \text{ (d) Write skew. Initially acct} \_1 = \text{acct} \_2 = 60. \\
& \quad \text{ T}_1 \rightarrow \text{ read}(x, 1) \\
& \quad \text{ acct} := \text{acct} + 50 \\
& \quad \text{ T}_2 \rightarrow \text{ write}(x, 1) \\
& \quad \text{ acct} := \text{acct} + 25
\end{align*}\]

\[\begin{align*}
& \quad \text{ (b) Lost update.} \\
& \quad \text{ (c) Long fork.} \\
& \quad \text{ T}_1 \rightarrow \text{ write}(x, 1) \\
& \quad \text{ acct} := \text{acct} + 50 \\
& \quad \text{ T}_2 \rightarrow \text{ write}(x, 1) \\
& \quad \text{ acct} := \text{acct} + 25
\end{align*}\]

We write \( T \not\triangleright \text{VIS} \) \( S \) and \((T, S) \in \text{VIS}\) interchangeably, and similarly for other relations. For \( \mathcal{H} = (T, S, \text{VIS}) \) we shorten \((T, S, \text{VIS}, \text{CO})\) to \((\mathcal{H}, \text{VIS}, \text{CO})\). In terms of the SI concurrency-control algorithm sketched in §1, \( T \not\triangleright \text{VIS} \rightarrow S \) means that the writes done by the transaction \( T \) are included into the snapshot taken by the transaction \( S; T \rightarrow \text{CO} \leftrightarrow S \) means that \( T \) commits earlier than \( S \). The constraint \( \text{VIS} \subseteq \text{CO} \) ensures that the snapshot taken by a transaction may only include previously committed transactions. SI or serializability allow those histories that can be extended to an abstract execution satisfying certain \textit{consistency axioms} from Figure 1, which specify the corresponding guarantees about transaction processing.

\textbf{Definition 4.} \textit{The sets of executions and histories allowed by (strong session) SI and serializability are:}

\[
\begin{align*}
\text{ExecSI} & = \{ \mathcal{X} \mid \mathcal{X} = \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{PREFIX} \land \text{NOCONFLICT} \}; \\
\text{ExecSER} & = \{ \mathcal{X} \mid \mathcal{X} = \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{TOTALVIS} \}; \\
\text{HistSI} & = \{ \mathcal{H} \mid \exists \text{VIS}, \text{CO}, (\mathcal{H}, \text{VIS}, \text{CO}) \in \text{ExecSI} \}; \\
\text{HistSER} & = \{ \mathcal{H} \mid \exists \text{VIS}, \text{CO}, (\mathcal{H}, \text{VIS}, \text{CO}) \in \text{ExecSER} \}.
\end{align*}
\]

We now explain the axioms in Figure 1, as well as anomalies that SI allows or disallows; the latter are summarised in Figure 2. We use the following notation. For a set \( A \) and a total order \( R \subseteq A \times A \), we let \( \max_R(A) \) be the element \( a \in A \) such that \( \forall b \in A. a = b \lor (b, a) \in R \); if \( A = \emptyset \), then \( \max_R(A) \) is undefined. In the following, the use of \( \max_R(A) \) in an expression implicitly assumes that it is defined. We define \( \min_R(A) \) similarly. For a relation \( R \subseteq A \times A \) and an element \( a \in A \), we let \( R^{-1}(a) = \{ b \mid (b, a) \in R \} \).

We define the sequential composition of relations \( R_1 \) and \( R_2 \) as

\[
R_1 \circ R_2 = \{ (a, b) \mid \exists c. (a, c) \in R_1 \land (c, b) \in R_2 \}.
\]

We write \_ for a value that is irrelevant and implicitly existentially quantified.

The \textit{INT} and \textit{EXT} axioms in Figure 1 ensure that a transaction reads from a snapshot of object states and its own writes. The \textit{internal consistency axiom} \textit{INT} ensures that a read event \( e \) on an object \( x \) returns the same value as the last write to or a read from \( x \) preceding \( e \) in the same transaction. If a read is not preceded in the same transaction by an operation on the same object, then its value is determined in terms of writes by other transactions using the \textit{external consistency axiom} \textit{EXT}. For \( T = (E, po) \), we let \( T \vdash \text{write}(x, n) \) if \( T \) writes to \( x \) and the last value written is \( n \):

\[
\text{op}(\maxpo\{ e \mid \text{op}(e) = \text{write}(x, \_ \}) = \text{write}(x, n).
\]
We let \( T \vdash \text{read}(x, n) \) if \( T \) reads from \( x \) before writing to it and \( n \) is the value returned by the first such read:
\[
\text{op}(\min_{\text{op}}\{e \mid \text{op}(e) = \langle x, \_ \rangle\}) = \text{read}(x, n).
\]
We also let \( \text{WriteTx}_x = \{ T \mid T \vdash \text{write}(x, \_\) \}. \) Then Ext ensures that, if a transaction \( T \) reads an object \( x \) before writing to it, then the value read is determined by the transactions that are included into \( T \)'s snapshot according to \( \text{VIS} \) and that wrote to \( x \); \( T \) reads the value written by the transaction from this set that committed last according to \( \text{CO} \). For simplicity, we consider only executions where the above set is always non-empty; this can be ensured by introducing a special transaction that writes initial values of all objects. The executions in Figures 2(a) and 2(b) satisfy Ext.

Our specification determines the snapshot that a transaction reads from based on an arbitrary visibility relation and does not require the snapshot to be "latest"; this is similar to so-called \textit{generalised SI} \cite{18}. However, following strong session SI [13, 14], the \textsc{session} axiom requires the snapshot to include the effects of all preceding transactions in the same session. For example, in the execution in Figure 2(a), the session order between \( T_1 \) and \( T_2 \) induces a visibility edge according to \textsc{session}.

The \textsc{prefix} axiom ensures that, if the snapshot taken by a transaction \( T \) includes a (committed) transaction \( S \), then this snapshot also includes all transactions that committed before \( S \). Note that \textsc{prefix} and the property \( \text{VIS} \subseteq \text{CO} \) in Definition 4 imply that \( \text{VIS} \) is transitive. \textsc{prefix} disallows the long fork anomaly shown in Figure 2(c), which is allowed by some weakening of SI (as such as parallel SI \cite{32}). There transactions \( T_1 \) and \( T_2 \) concurrently write to objects \( x \) and \( y \). Transaction \( T_3 \) sees the write by \( T_1 \), but not the write by \( T_2 \); conversely, transaction \( T_4 \) sees the write by \( T_2 \), but not the write by \( T_1 \). Thus, from the perspectives of \( T_3 \) and \( T_4 \), the writes of \( T_1 \) and \( T_2 \) happen in different orders. \textsc{prefix} disallows any execution with the history in Figure 2(c), because in such an execution \( T_1 \) and \( T_2 \) have to be related by CO one way or another; but then by \textsc{prefix}, either \( T_1 \) has to observe the write to \( x \) or \( T_3 \) has to observe the write to \( y \).

The axioms explained so far do not prevent the lost update anomaly, illustrated by the execution in Figure 2(b). This execution could arise from the code in the figure that uses transactions \( T_1 \) and \( T_2 \) to make deposits into an account. The two transactions read the initial balance of the account and concurrently modify it. The two transactions read the initial balance of the account and concurrently modify it. The two transactions read the initial balance of the account and concurrently modify it.

The set \textsc{HistSI} (Definition 4) defined using the consistency axioms explained so far is exactly the one produced by the SI concurrency-control algorithm \cite{10}. The axioms allow the execution in Figure 2(d) with the characteristic SI anomaly of \textit{write skew} (§1), disallowed by serializability. We formalise the latter by the axiom TOTALVIS, which requires visibility to totally order all transactions. Then the axioms \textsc{int} and \textsc{ext} ensure that the transactions are processed according to the usual sequential semantics. We thus have \textsc{HistSER} \subseteq \textsc{HistSI}.

### 3. DEPENDENCY GRAPHS

From an abstract execution we can extract several kinds of dependencies between its transactions, which are used in consistency model specifications in the style of Adya et al. \cite{2, 3}.

\textbf{Definition 5.} Let \( \mathcal{X} = (\mathcal{H}, \text{VIS}, \text{CO}) \) be an execution. For \( x \in \text{Obj} \), we define the following relations on \( \mathcal{T}_x \):

- \textbf{read dependency:} \( T \xrightarrow{\text{rw}_X(x)} S \iff S \vdash \text{read}(x, \_) \land T = \max_{\text{CO}}(\text{VIS}^{-1}(S) \cap \text{writeTx}_x); \)
- \textbf{write dependency:} \( T \xrightarrow{\text{ww}_X(x)} S \iff T \xrightarrow{\text{CO}} S \land T, S \in \text{WriteTx}_x; \)
- \textbf{anti-dependency:} \( T \xrightarrow{\text{rw}_X(x)} S \iff T \neq S \land \exists T', T' \xrightarrow{\text{rr}_X(x)} T \land T' \xrightarrow{\text{ww}_X(x)} S. \)

Informally, \( T \xrightarrow{\text{rw}_X(x)} S \) means that \( S \) reads \( T \)'s write to \( x \) (cf. the \textsc{ext} axiom in Figure 1); \( T \xrightarrow{\text{ww}_X(x)} S \) means that \( S \) overwrites \( T \)'s write to \( x \); \( T \xrightarrow{\text{rw}_X(x)} S \) means that \( S \) overwrites the write to \( x \) read by \( T \). For example, the dependencies of the executions in Figures 2(b), 2(d) and 2(c) are shown there with bold arrows (keep in mind that the pictures omit a special initialisation transaction). We often abuse notation and use the symbol \( \text{WR}_X \) to also denote the relation \( \bigcup_{x \in \text{Obj}} \text{WR}_X(x) \subseteq \mathcal{T}_X \times \mathcal{T}_X \), and similarly for \( \text{WW}_X \) and \( \text{RW}_X \).

A key goal of this paper is to characterise SI solely in terms of dependencies: we want to determine whether SI allows a given history by looking for appropriate dependencies between its transactions rather than visibility and commit orders, as in Definition 4. To this end, we extend histories to \textit{dependency graphs} (aka direct serialization graphs) \cite{2}, which include relations representing the dependencies.

\textbf{Definition 6.} A \textit{dependency graph} is a tuple \( \mathcal{G} = (\mathcal{T}, \text{SO}, \text{WR}, \text{WW}, \text{RW}), \) where \( (\mathcal{T}, \text{SO}) \) is a history and

- \textbf{WR} : \text{Obj} \to 2^{\mathcal{T} \times \mathcal{T}} is such that:

  - \( \forall T, S \in \mathcal{T}, \forall x, T \xrightarrow{\text{wr}_X(x)} S \implies \exists n. T \neq S \land T \vdash \text{write}(x, n) \land S \vdash \text{read}(x, n); \)

  - \( \forall S \in \mathcal{T}, \forall x, S \vdash \text{read}(x, \_) \implies \exists T, T \xrightarrow{\text{wr}_X(x)} S; \)

  - \( \forall T, T', S \in \mathcal{T}, \forall x, (T \xrightarrow{\text{wr}_X(x)} S \land T' \xrightarrow{\text{wr}_X(x)} S) \implies T = T'; \)

- \textbf{WW} : \text{Obj} \to 2^{\mathcal{T} \times \mathcal{T}} is such that for every \( x \in \text{Obj} \), \( \text{WW}(x) \) is a total order on the set \( \text{WriteTx}_x; \)

- \textbf{RW} : \text{Obj} \to 2^{\mathcal{T} \times \mathcal{T}} is derived from \text{WR} and \text{WW} as in Definition 5.

\textbf{Proposition 7.} For any \( \mathcal{X} \in \text{ExecSI} \), \( \text{graph}(\mathcal{X}) = (\mathcal{T}, \text{SO}_X, \text{WR}_X, \text{WW}_X, \text{RW}_X) \) is a dependency graph.

Note that the constraints on \text{WR} in Definition 6 ensure that it uniquely determines the values read by transactions. For \( \mathcal{H} = (\mathcal{T}, \text{SO}) \) we write \( (\mathcal{H}, \text{WR}, \text{WW}, \text{RW}) \) for \( (\mathcal{T}, \text{SO}, \text{WR}, \text{WW}, \text{RW}) \).

We write \( T \models \text{int} \) if a set of transactions \( T \) satisfies the internal consistency axiom \text{int} in Figure 1. (Strong session) serializability can be characterised by the set of acyclic dependency graphs with internally consistent transactions \cite{2}.

\textbf{Theorem 8.} Let
\[
\text{GraphSER} = \{ \mathcal{G} \mid (\mathcal{T}_0 \models \text{int}) \land ((\text{SO}_\mathcal{G} \cup \text{WR}_\mathcal{G} \cup \text{WW}_\mathcal{G} \cup \text{RW}_\mathcal{G}) \text{ is acyclic}) \}.
\]
For a set $T$ and a relation $R \subseteq T \times T$ let $R^? = R \cup \{(T, T) \mid T \in T\}$. We show that (strong session) SI is characterised by dependency graphs that contain only cycles with at least two adjacent anti-dependencies.

**Theorem 9.** Let $\text{GraphSI} = \{ G \mid \langle T_0 \models \text{INT} \rangle \wedge ((\langle SO_G \cup WR_G \cup WW_G \rangle : RW_G ?) \text{ is acyclic})\}$. Then

$\text{HistSI} = \{ H \mid \exists \text{WR}, \text{WW}, \text{RW} . (H, \text{WR}, \text{WW}, \text{RW}) \in \text{GraphSI}\}$.

According to the theorem, to determine whether a particular history is allowed by SI, we can look for dependencies that extend it to a graph in GraphSI. As we demonstrate in §5 and §6, this way of defining SI is particularly suitable for developing static analyses for this consistency model. The history in Figure 2(d) is allowed by SI, and indeed the dependency graph shown in the figure contains only cycles with two adjacent anti-dependencies (e.g., $T_1 \xrightarrow{\text{RW}} T_2 \xrightarrow{\text{RW}} T_1$). In contrast, the histories in Figures 2(b) and 2(c) are not allowed by SI, and they cannot be extended to graphs where every cycle has at least two adjacent anti-dependencies. In particular, the graphs shown in the figures contain cycles without these: e.g., $T_1 \xrightarrow{\text{WW}} T_2 \xrightarrow{\text{RW}} T_1$ in Figure 2(b) and $T_1 \xrightarrow{\text{WR}} T_3 \xrightarrow{\text{RW}} T_2 \xrightarrow{\text{WR}} T_4 \xrightarrow{\text{RW}} T_1$ in Figure 2(c).

To prove Theorem 9, we prove a slightly stronger result, showing that we can establish a correspondence between executions in ExecSI and graphs in GraphSI that preserves histories and dependencies.

**Theorem 10.**

(i) **Soundness:** $\forall G \in \text{GraphSI} \exists X \in \text{ExecSI}. \text{graph}(X) = G$.

(ii) **Completeness:** $\forall X \in \text{ExecSI}. \text{graph}(X) \in \text{GraphSI}$.

As we explain in §7, the easier completeness direction of this theorem actually follows from existing results [19]. Our main technical contribution is the more challenging proof of the soundness direction, which is required for the static analyses that we propose (§5 and §6). We present this proof first.

The main challenge is to construct a total commit order in the desired execution $X$ from the dependencies given by $G$ while satisfying the SI axioms (Definition 4). We do this incrementally; at intermediate stages of the construction we get structures similar to abstract executions, but where the commit order can be partial.

**Definition 11.** A tuple $P = (T, \text{SO}, \text{VIS}, \text{CO})$ is a pre-execution if it satisfies all the conditions of Definition 3, except CO is a strict partial order that may not be total. We let PreExecSI be the set of pre-executions satisfying the SI axioms (Figure 1):

$\text{PreExecSI} = \{ P \mid P \models \text{INT} \wedge \text{EXT} \wedge \text{SESSION} \wedge \text{PREFIX} \wedge \text{NoConflict}\}$.

Thus, an execution is a pre-execution whose commit order is total. In the following, we apply the graph function of §3 also to pre-executions; for $P \in \text{PreExecSI}$ graph($P$) is indeed a dependency graph.

We first obtain auxiliary results that, given a dependency graph $G = (T, \text{SO}, \text{WR}, \text{WW}, \text{RW}) \in \text{GraphSI}$, allow us to construct a pre-execution $P = (T, \text{SO}, \text{VIS}, \text{CO}) \in \text{PreExecSI}$ such that graph($P$) = $G$. Later we show how to extend $P$ to a desired execution $X \in \text{ExecSI}$. We start by restating the requirements on the pre-execution $P$ in a way more suitable for guiding its construction; these are given by the system of inequalities in Figure 3. First, to ensure graph($P$) = $G$, by Definition 5 at the very least we must have $\text{WR} \cup \text{WW} \subseteq \text{VIS}$. For $P$ to satisfy the Session axiom we must also have $\text{SO} \subseteq \text{VIS}$. These two observations motivate (S1). This inequality also implies that $P$ satisfies NoConflict, since according to Definition 5, WW is total over transactions that write to a given object. Inequality (S2) is equivalent to Prefix, and inequality (S3) states a relationship between VIS and CO inherited by Definition 11 from Definition 3. Inequality (S4) requires $\text{CO}$ to be transitive: (S2) and (S3) ensure that so is VIS.

As we now explain, (S5) ensures the axiom Ext. Consider the dependency graph $G \in \text{GraphSI}$ in Figure 4, which we use as our running example in this section and in §5. Its transactions could arise from the programs, also shown in the figure, that make a transfer between two accounts and query their balances or the sum thereof. The transactions arising from 1000kp1 and 1000kp2 see the initial state of the database, while the transactions arising from 1000kp411 see its state in the middle of a transfer. The VIS and CO relations shown by the solid arrows give a pre-execution satisfying inequalities (S1)-(S4) and, in fact, all the SI axioms. Suppose we want to construct a pre-execution with a bigger CO by adding an edge $T \xrightarrow{\text{CO}} S'$ (shown dotted). Since $S' \xrightarrow{\text{VIS}} S$, for the resulting pre-execution to satisfy Prefix we also need to add an edge $T \xrightarrow{\text{VIS}} S$. But then the pre-execution violates Ext: $S$ sees the write to acct2 by $T$, but reads the value from the initialisation transaction (elided from Figure 4) that writes 0 to acct2 and precedes $T$ in CO and WW; the latter fact is witnessed by the edge $S \xrightarrow{\text{RW}} T$. On the other hand, adding an edge $S' \xrightarrow{\text{CO}} T$ (shown dashed), which belongs to VIS; RW, does not violate Ext. This example illustrates a general pattern. First, as the following lemma shows, (S5) must hold in any SI execution.

**Lemma 12.** $\forall X \in \text{ExecSI}. \text{VIS}_X \subseteq \text{CO}_X$.

Conversely, the system of inequalities in Figure 3 can be used to ensure that a pre-execution $P = (T, \text{SO}, \text{VIS}, \text{CO})$ satisfies Ext and has other desired properties.

**Lemma 13.** Let $G = (T, \text{SO}, \text{WR}, \text{WW}, \text{RW})$ be a dependency graph such that $T \models \text{INT}$ and VIS, CO $\subseteq T \times T$ be acyclic relations satisfying the system of inequalities in Figure 3. Then $G \in \text{GraphSI}$.

![Figure 3: Requirements on a pre-execution $P = (H, \text{VIS}, \text{CO})$ constructed from a dependency graph $G = (H, \text{WR}, \text{WW}, \text{RW})$.](image)
3. Then \( \mathcal{P} = (T, SO, VIS, CO) \) is a pre-execution such that \( \mathcal{P} \in \text{PreExecSI} \) and graph(\( \mathcal{P} \)) = \( \mathcal{G} \).

The proof of Lemma 12 depends on the following characterisation of anti-dependencies in terms of visibility edges.

**Proposition 14.**
\[
\forall X \in \text{ExecSI}. \forall T, S \in T_{\mathcal{X}}. \exists x. S \vdash \text{read}(x, \_). T \leftarrow S \equiv \neg T \land \text{write}(x, \_). \neg S \land (T \leftarrow X \rightarrow S).
\]

Informally, if we had \( S \leftarrow X \rightarrow T \) and \( T \leftarrow VIS \rightarrow S \), then \( S \) would have to read a value of \( x \) at least as up-to-date as that written by \( T \), contradicting the definition of \( RW_x \).

**Proof of Lemma 12.** Consider \( X \in \text{ExecSI} \) and \( T, S', S' \in T_{\mathcal{X}} \) such that \( S' \leftarrow VIS \rightarrow S \leftarrow RW \rightarrow T \). If \( T = S' \), then \( S' \leftarrow VIS \rightarrow S \leftarrow RW \rightarrow T \), contradicting Proposition 14. If \( T \leftarrow CO \rightarrow S' \) then by Proposition 14, then by Prefix we get \( T \leftarrow VIS \rightarrow S \), contradicting Proposition 14. Then, since \( CO \) is total, we must have \( S' \leftarrow CO \rightarrow T \). \( \square \)

**Proof of Lemma 13.** We only prove that \( \mathcal{P} \models \text{Ext} \) and \( WR_\mathcal{P} = WR \), discharging the other obligations is straightforward. Consider \( S \in T \) such that \( S \leftarrow \text{read}(x, n) \). Then there exists a unique \( T' \) such that \( T' \leftarrow \text{read}(x, n) \rightarrow S \). Let \( T = \text{maxCO}(\text{VIS}^{-1}(S) \cap \text{WriteTx}_x) \). This is defined because: \( CO \) is acyclic; by (S1) and (S3) we have \( WW \subseteq CO \), so that \( CO \) is total over \( \text{WriteTx}_x \); and by (S1) we have \( WR \subseteq VIS \), so that \( T' \in \text{VIS}^{-1}(S) \cap \text{WriteTx}_x \). We now show that \( T = T' \), which entails the required.

Assume the contrary: \( T \neq T' \). We have \( T, T' \in \text{VIS}^{-1}(S) \cap \text{WriteTx}_x \). Hence, \( T \) and \( T' \) are related by \( WW \). Since \( WW \subseteq CO \), they are related in the same way by the acyclic \( CO \). Then by the definition of \( T \) we must have \( T' \leftarrow CO \rightarrow T \) and \( T \leftarrow CO \rightarrow T' \). From the latter and \( T' \leftarrow WR \rightarrow S \) we get \( S \leftarrow RW \rightarrow T \). But \( T \leftarrow VIS \rightarrow S \), so from (SS) we get \( T \leftarrow CO \rightarrow T \). This contradicts the assumption that \( CO \) is acyclic. Hence, we must have \( T = T' \). \( \square \)

According to Lemma 13, to construct a desired pre-execution \( \mathcal{P} = (T, SO, VIS, CO) \in \text{PreExecSI} \) from a dependency graph \( \mathcal{G} = (T, SO, WR, WW, RW) \), it is sufficient to find a solution to the system of inequities in Figure 3 in terms of acyclic relations \( VIS \) and \( CO \). This is not completely trivial because of the recursive nature of the inequalities: according to them, adding more edges into \( VIS \) forces adding more edges into \( CO \) and vice versa, increasing the risk of tying a cycle. Our insight is to look for the solution that is smallest and, hence, least likely to contain cycles. The following lemma gives a closed form for this solution. In anticipation of using the lemma when extending a pre-execution to an execution, we state it in a generalised form that gives the smallest solution where \( CO \) contains at least a given set of edges \( R \). We use \(+\) and \(*\) to denote the transitive closure and the transitive and reflexive closure of a given relation.

**Lemma 15.** Let \( \mathcal{G} = (T, SO, WR, WW, RW) \) be a dependency graph. For any \( R \subseteq T \times T \), the relations
\[
\begin{align*}
\text{VIS} &= (\{(SO \cup WR \cup WW) \cap RW \}^* \cap \{ VIS \}^+) + (SO \cup WR \cup WW); \\
\text{CO} &= (\{(SO \cup WR \cup WW) \cap RW \}^+ \cup \{ VIS \}^+) \cup \{ CO \}
\end{align*}
\]
are a solution to the system of inequalities in Figure 3. They also are the smallest solution to the system for which \( CO \supseteq R \) for any other solution \( \{ VIS', CO' \} \) with \( CO' \supseteq R \) we have \( VIS \subseteq VIS' \) and \( CO \subseteq CO' \).

In particular, for \( R = \emptyset \), Lemma 15 gives the smallest solution \( (VIS_0, CO_0) \) to the system of inequalities in Figure 3.

If \( \mathcal{G} \) is GraphSI, then \( CO_0 \) is acyclic, and by (S3), so is \( VIS_0 \) (in fact, Lemma 15 is our motivation for defining GraphSI the way we did). Hence, by Lemma 13, \( \mathcal{P}_0 = (T, SO, VIS_0, CO_0) \) is a pre-execution such that \( \mathcal{P}_0 \in \text{PreExecSI} \) and graph(\( \mathcal{P}_0 \)) = \( \mathcal{G} \), which is what we originally set out to construct. We now proceed to prove Theorem 10(i) by extending the pre-execution \( \mathcal{P}_0 \) to an execution \( \mathcal{X} \in \text{ExecSI} \).

**Proof of Theorem 10(i).** Assume \( \mathcal{G} = (T, SO, WR, WW, RW) \in \text{GraphSI} \). To construct \( \mathcal{X} \), we define a sequence of pre-executions \( \{ \mathcal{P}_i = (T, SO, VIS_i, CO_i) \}_{n=0}^\infty \) for some \( n \geq 0 \) and let \( \mathcal{X} = \mathcal{P}_n \). The sequence is such that \( VIS_i \subseteq VIS_{i+1} \) and \( CO_i \subseteq CO_{i+1} \) for \( i = 0, \ldots, (n-1) \); furthermore, \( CO_n \) is total, so that \( \mathcal{P}_n \) is an execution. That is, on every step of our construction we add edges to the commit order until it becomes total. Each pair \( (VIS_i, CO_i) \) gives an acyclic solution to the system in Figure 3. Then by Lemma 13, \( \mathcal{P}_i \in \text{PreExecSI} \) and graph(\( \mathcal{P}_i \)) = \( \mathcal{G} \). In particular, \( \mathcal{P}_n \in \text{PreExecSI} \) and graph(\( \mathcal{P}_n \)) = \( \mathcal{G} \), as required.
5. TRANSACTION CHOPPING UNDER SI

In this section, we exploit our characterisation of SI in terms of dependency graphs to derive a static analysis that checks when transactions in an application executing under SI can be chopped [30] into sessions of smaller transactions without introducing new behaviours (the sessions are also called chains in this context [38]). To this end, the analysis must check that any SI execution of the application with chopped transactions can be spliced into an SI execution that has the same operations as the original one, but where all operations from each session are executed inside a single transaction. We first establish a dynamic chopping criterion that checks whether a single SI execution, represented by a dependency graph, is spliceable. From this we then derive a static analysis that checks whether this is the case for all executions produced by a given chopped application.

For a history $\mathcal{H}$, let $\approx_{\mathcal{H}} = \equiv_{\mathcal{H}} \cup \{(T, T) \mid T \in \mathcal{H}\}$ be the equivalence relation grouping transactions from the same session. We let $\bar{T}_{\mathcal{H}}$ be the result of splicing all transactions in the session to which $T$ belongs in $\mathcal{H}$ into a single transaction: $\bar{T}_{\mathcal{H}} = (E, po)$, where $E = \langle \{ | E_S \mid S \approx_{\mathcal{H}} T \} \rangle$ and

\[
\text{po} = \{(e, f) \mid (\exists S', e \in E_S \land e \overset{\text{po}}{\rightarrow} f \land S \approx_{\mathcal{H}} T) \land (\exists S', e \in E_S \land f \in E_S' \land S' \overset{\text{po}}{\rightarrow} S'') \approx_{\mathcal{H}} T\}.
\]

We let splice$(\mathcal{H})$ be the history resulting from splicing all sessions in a history $\mathcal{H}$: splice$(\mathcal{H})$ = $\{ \{T_{\mathcal{H}} \mid T \in \mathcal{T}_{\mathcal{H}} \}, \emptyset \}$. A dependency graph $G \in \text{GraphSI}$ is spliceable if there exists a dependency graph $G' \in \text{GraphSI}$ such that $H_{G'} = \text{splice}(H_G)$. For a dependency graph $G$, we let $\approx_{\mathcal{G}} = \approx_{\mathcal{H}_{G}}$ and $\bar{T}_{\mathcal{G}} = \bar{T}_{\mathcal{H}_{G}}$.

For example, the graph $G_1$ in Figure 4 is not spliceable, because splice$(G_1) \notin \text{HistSI}$: informally, $S_{G_1}$ observes the write by $T_{G_1}$ to acct1, but not its write to acct2. On the other hand, let $G_2$ be the graph obtained by removing the transactions $S$ and $S'$ from $G_1$. Then $G_2$ is spliceable, as witnessed by the graph $G_3 \in \text{GraphSI}$ with $\mathcal{H}_{G_2}$ = splice$(\mathcal{H}_{G_3})$ and only the edges $\bar{T}_{G_3} = \bar{T}_{G_2}$ and $S'_{G_3}$. Given a dependency graph $G$, we let the dynamic chopping criterion corresponding to $G$ be the graph $(\text{DCG}(G))$ obtained by removing $\approx_{\mathcal{G}}$, $\text{WW}_{\mathcal{G}}$, and $\text{RW}_{\mathcal{G}}$ edges between transactions related by $\approx_{\mathcal{G}}$, and by extending $\approx$ with edges in the reverse of the session order: $\equiv_{\mathcal{G}}^{-1}$. We refer to the latter edges as predecessor edges, to those in $\equiv_{\mathcal{G}}$ as successor edges, and to those in $(\text{WR} \cup \text{WW} \cup \text{RW}) \setminus \approx_{\mathcal{G}}$ as conflict edges. A cycle in a chopping graph $(\text{DCG}(G))$ is critical if: (i) it does not contain two occurrences of the same vertex; (ii) it contains at most three edges of the form “conflict, predecessor, conflict”; and (iii) at most two edges of the form $(\text{WR} \cup \approx_{\mathcal{G}})$ or $(\text{WW} \cup \approx_{\mathcal{G}})$.

Our dynamic chopping criterion is as follows.

**Theorem 16.** For $G \in \text{GraphSI}$, if $(\text{DCG}(G))$ contains no critical cycles, then $G$ is spliceable.

For example, the above graph $G_2$ (Figure 4) contains no critical cycles, and the graph $G_1$ contains a critical cycle $T' \xrightarrow{\text{RW}_{G_1}} S' \xrightarrow{\equiv_{G_1}} S \xrightarrow{\text{RW}_{G_1}} T \xrightarrow{\equiv_{G_1}} T'$.

To prove Theorem 16, we exhibit a particular dependency graph $(\text{splice}(G))$ such that splice$(\mathcal{G}) \in \text{GraphSI}$ and $H_{\text{splice}(G)} = \text{splice}(H_G)$. We define read dependances $\text{WR}_{\text{splice}(G)}$ by lifting those in $\text{WR}_G$ to spliced transactions:

\[
\forall T, S \in \mathcal{T}_G, \forall x \in \text{Obj}, \bar{T}_{\text{splice}(G)}(x) \xrightarrow{\text{WR}_{\text{splice}(G)}(x)} S_{\text{splice}(G)} \iff \bar{T}_{\overline{\text{splice}(G)}}(x) \xrightarrow{\text{WR}(x)} \bar{T}_G \neq S_{\text{splice}(G)} \land \bar{T}_G \neq S_{\text{splice}(G)} \Rightarrow S.
\]

We define $\text{WW}_{\text{splice}(G)}$ similarly and derive $\text{RW}_{\text{splice}(G)}$ from $\text{WR}_{\text{splice}(G)}$ and $\text{WW}_{\text{splice}(G)}$ as in Definition 5. As the following lemma shows, $\text{RW}_{\text{splice}(G)}$ defined in this way can be decomposed into a form similar to (2).

**Lemma 17.** Let $G \in \text{GraphSI}$ be such that $(\text{DCG}(G))$ contains no critical cycles. Then

\[
\forall T, S \in \mathcal{T}_G, \forall x \in \text{Obj}, \bar{T}_{\text{splice}(G)}(x) \xrightarrow{\text{RW}_{\text{splice}(G)}(x)} S_{\text{splice}(G)} \Rightarrow \bar{T}_{\overline{\text{splice}(G)}}(x) \xrightarrow{\text{RW}(x)} \bar{T}_G \neq S_{\text{splice}(G)} \land \bar{T}_G \neq S_{\text{splice}(G)} \Rightarrow S.
\]

To prove Theorem 16, we assume splice$(G) \notin \text{GraphSI}$ and use Theorem 9 to obtain a cycle in $(\text{WR}_{\text{splice}(G)} \cup \text{WW}_{\text{splice}(G)})$;
We now derive a static analysis from Theorem 16. Assume a set of programs \( P = \{ P_1, P_2, \ldots \} \), each defining the code of sessions resulting from chopping the code of a single transaction. We leave the precise syntax of the programs unspecified, but assume that each \( P_i \) consists of \( k_i \) \( \textit{program pieces} \), defining the code of the transactions in the sessions. We further assume that we are given the sets \( R_i \) and \( W_i^j \) of all objects that can respectively be read and written by the \( j \)-th piece of \( P_i \). For example, the program \( \text{transfer} \) in Figure 4 consists of two pieces; the first one has the read and write sets equal to \{acct1\} and the second, to \{acct2\}. The program \( \text{lookup1} \) consists of a single piece with the read set \{acct1\} and the write set \emptyset.

Following Shasha et al. [30], we make certain assumptions about the way clients execute programs. We assume that, if a transaction initiated by a program piece aborts, it will be resubmitted repeatedly until it commits, and, if a piece is aborted due to system failure, it will be restarted. We also assume that the client does not abort transactions explicitly.

A history \( \mathcal{H} \) \textit{can be produced} by the programs \( P \), if there is a one-to-one correspondence between every session in \( \mathcal{H} \) and a program \( P_i \in P \) whose read and write sets cover the sets of objects read or written by the corresponding transactions in the session. For example, the history in Figure 4 can be produced by the programs in the figure. The chopping defined by the programs \( P \) is \textit{correct} if every dependency graph \( G \in \text{GraphSI} \), where \( \mathcal{H}_G \) can be produced by \( P \), is splicable.

We check the correctness of \( P \) using its \textit{static chopping graph} \( \text{SCG}(P) \). It is a directed graph whose nodes are pairs of indices identifying the pieces in \( P \) \( \{ (i, j) \mid i = 1..|P|, j = 1..k_i \} \). We have an edge \((i_1, j_1), (i_2, j_2)\) if and only if one of the following holds: \( i_1 = i_2 \) and \( j_1 < j_2 \) (a \textit{successor} edge); \( i_1 = i_2 \) and \( j_1 > j_2 \) (a \textit{predecessor} edge); \( i_1 \neq i_2 \) and \( W_{i_1}^j \cap R_{i_2}^j \neq \emptyset \) (a \textit{dependency} edge); \( i_1 \neq i_2 \) and \( W_{i_1}^j \cap W_{i_2}^j \neq \emptyset \) (a \textit{dependency} edge); or \( i_1 \neq i_2 \) and \( R_{i_1}^j \cap W_{i_2}^j \neq \emptyset \) (an \textit{anti-dependency} edge). The notion of a critical cycle introduced above for dynamic graphs is also applicable to static ones. The edge set of a static graph \( \text{SCG}(P) \) over-approximates the edge sets of dynamic graphs \( \text{DCG}(G) \) corresponding to dependency graphs \( G \) produced by the programs \( P \). From this observation and Theorem 16 we easily get our static analysis.

\textbf{Corollary 18.} The chopping defined by \( P \) is correct if \( \text{SCG}(P) \) contains no critical cycles.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.pdf}
\caption{The static chopping graph of the programs \{\textit{transfer,lookupAll}\} from Figure 4. Dashed boxes group program pieces into sessions.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.pdf}
\caption{The static chopping graph of the programs \{\textit{transfer,lookup1,lookup2}\} from Figure 4.}
\end{figure}

In Figure 5 we show the static chopping graph of the programs \{\textit{transfer,lookupAll}\}, which contains a critical cycle:
\[
\begin{align*}
\text{(var1 = acct1)} \xrightarrow{\text{RW}} & (acct1 = acct1 - 100) \\
\text{(acct2 = acct2 + 100)} \xrightarrow{\text{WR}} & (var2 = acct2) \\
\text{(var2 = acct2)} \xrightarrow{\text{S}} & (\text{var1 = acct1}).
\end{align*}
\]

In fact, since the dependency graph in Figure 4 is not spliceable, the chopping defined by the above programs is incorrect. In Figure 6 we show the static chopping graph of the programs \{\textit{transfer,lookup1,lookup2}\}. This graph contains no critical cycles, and hence, the chopping defined by these programs is correct: they behave the same as when \textit{transfer} is implemented by a single transaction.

Our SI characterisation is instrumental in deriving the above static analysis due to the ease of splicing a dependency graph (cf. (2)). As we explain in [11, §B], splicing abstract executions directly would be problematic.

In [11, §B], we also show that the conditions on chopping required by Corollary 18 are laxer than those of the analysis for serializability [30], but stricter than those for parallel SI [12]. In particular, this implies that the classical transaction chopping analysis for serializability is also sound for SI. This result is non-trivial: the correctness of a chopping requires that the set of histories produced by the chopped program be included into the set of histories produced by the original program. Enlarging both sets when switching from serializability to SI may not preserve the inclusion.

\section{Robustness Criteria for SI}

We now consider another type of a static analysis that checks whether an application is \textit{robust} against weakening consistency: executing it under a weak consistency model produces the same client-observable behaviour as executing it under a stronger one.

\subsection{Robustness against SI}

We first show that our SI characterisation allows deriving a variant of an existing analysis that checks whether an application executing under SI behaves the same as when executing under serializability [19]\footnote{Our analysis does not take into account the notion of a \textit{vulnerable} dependency from [19].} (\textit{robustness against SI}). For this the analysis checks that the application code may produce no histories in \text{HistSI} \setminus \text{HistSER}. Like for transaction chopping (§5), we first establish a \textit{dynamic robustness criterion} that checks whether a single execution, represented by a dependency graph, is in \text{GraphSI} \setminus \text{GraphSER}. This easily follows from Theorems 8 and 10.
Theorem 19. For any $G$, we have $G \in GraphSI \setminus GraphSER$ if and only if $T_G \models INT$, $G$ contains a cycle, and all its cycles have at least two adjacent anti-dependency edges.

Fekete et al. previously established a result corresponding to the “only if” direction of the above theorem [19]. The “if” direction strengthens their result by showing that the criterion in the theorem is complete for checking whether a given dependency graph is admitted by SI, but not serializability.

The dependency graph $G_3$ of the write skew anomaly in Figure 2(d) contains a cycle: $T_1 \xrightarrow{RW_3} T_2 \xrightarrow{RW_3} T_1$. Furthermore, it is easy to see that all its cycles have two adjacent anti-dependencies, so that $G_3 \in GraphSI \setminus GraphSER$. The dependency graph $G_4$ from Figure 4 does not contain any cycles, so that $G_4 \notin GraphSI \setminus GraphSER$; in fact, $G_4 \in GraphSER$. Finally, the dependency graph $G_1$ of the long fork anomaly in Figure 4 contains a cycle, but without two adjacent anti-dependencies; hence, $G_1 \notin GraphSI \setminus GraphSER$, and in fact, $G_1 \notin GraphSI$.

We can derive a static analysis from Theorem 19 similarly to how it was done in §5. Namely, the analysis assumes that the code of transactions in an application is defined by a set of programs $P$ with given read and write sets. Based on these sets, it constructs a static dependency graph, over-approximating possible dependencies that can exist in executions of the programs $P$. The analysis then checks that the graph has no cycles with at least two adjacent anti-dependency edges. By Theorem 19 this implies that the programs $P$ produce no histories in $HistSI \setminus HistSER$, and hence, the corresponding application is robust against SI. Note that the dependency graphs characterisation of consistency models greatly facilitates deriving the above static analysis, since the characterisations allow us to easily establish correspondences between executions on different models with the same histories.

6.2 Robustness against parallel SI towards SI

We now use our SI characterisation to derive a static analysis that checks whether an application executing under parallel SI [32] behaves the same as when executing under the classical SI (robustness against parallel SI towards SI). To specify parallel SI in the framework of §2, we drop the axiom PREFIX, while still requiring visibility to be transitive, a property that we refer to as TRANSVIS [10].

Definition 20. The sets of executions and histories allowed by parallel SI are:

$$ExecPSI = \{\mathcal{X} \mid \mathcal{X} \models INT \land Ext \land Session \land TRANSVIS \land NoConflict\};$$

$$HistPSI = \{\mathcal{H} \mid \exists VIS, CO, (\mathcal{H}, VIS, CO) \in \text{ExecSI}\}.$$

Note that this specification essentially does not use the commit order CO: according to NoConflict, its edges used in Ext are uniquely determined by VIS.

The axiom TRANSVIS ensures that transactions ordered by VIS are observed by others in this order. However, it allows two transactions unrelated by VIS to be observed in different orders; in particular, parallel SI allows the long fork anomaly of Figure 2(c), disallowed by the axiom PREFIX in SI.

Following [12, extended version, Lemma 14], we can give a characterisation of parallel SI in terms of dependency graphs.

Theorem 21. Let

$$GraphPSI = \{G \mid (T_G \models INT) \land ((SO_G \cup WR_G \cup WW_G)^+ : RW_G?) \text{ is irreflexive}\}.$$

Then

$$HistPSI = \{\mathcal{H} \mid \exists WR, WW, RW, (\mathcal{H}, WR, WW, RW) \in GraphPSI\}.$$

Thus, parallel SI is characterised by dependency graphs that contain only cycles with at least two anti-dependency edges. For example, consider the dependency graph $G_4$ in Figure 2(c). It is easy to see that all its cycles contain at least two anti-dependencies, and therefore $G_4 \in GraphPSI$. On the other hand, let $G_5$ be the dependency graph in Figure 2(b). The graph $G_5$ contains a cycle with exactly one anti-dependency ($T_1 \xrightarrow{WW_3} T_2 \xrightarrow{RW_3} T_1$), and therefore $G_5 \notin GraphPSI$. As a corollary of Theorems 9 and 21, we obtain a dynamic robustness criterion that checks whether a given dependency graph is in GraphPSI \ GraphSI.

Theorem 22. For any $G$, we have $G \in GraphPSI \setminus GraphSI$ if and only if $T_G \models INT$, $G$ contains at least one cycle with no adjacent anti-dependency edges, and all its cycles have at least two anti-dependency edges.

For example, we have already noted that in the dependency graph $G_4$ of the long fork anomaly all cycles have at least two anti-dependencies. Furthermore, $G_4$ also has a cycle with no adjacent anti-dependencies: $T_1 \xrightarrow{WR_3} T_3 \xrightarrow{WR_4} T_2 \xrightarrow{WR_4} T_4 \xrightarrow{RW_4} T_1$, so that $G_4 \in GraphPSI \setminus GraphSI$. The dependency graph $G_3$ of the write skew anomaly in Figure 2(d) contains only cycles with at least two adjacent anti-dependencies, so that $G_3 \notin GraphPSI \setminus GraphSI$; in fact, $G_3 \in GraphSI$. The dependency graph $G_3$ of the lost update anomaly contains a cycle with exactly one anti-dependency, so that $G_3 \notin GraphPSI \setminus GraphSI$; in fact, $G_3 \notin GraphPSI$.

From Theorem 22 it follows that the desired static analysis can check that the static dependency graph of an application contains no cycles where there are at least two anti-dependency edges and no two anti-dependency edges are adjacent.

7. RELATED WORK

Snapshot isolation was originally defined by an idealised algorithm formulated in terms of implementation-level concepts [6]. Since then there have been proposals of more declarative SI specifications [2, 10, 28], one of which [10] was our starting point (§2). However, these specifications are stated in terms of relations which make it challenging to obtain results such as transaction chopping and robustness analyses.

Fekete et al. [19] proposed the analysis for robustness against SI that we considered in §6.1. To this end, they have proved a fact roughly equivalent to our completeness result (Theorem 10(ii)), but they did not establish an analogue of our soundness result (Theorem 10(i)). The latter more challenging result is the one that is needed to obtain analyses for transaction chopping under SI and for robustness against parallel SI towards SI: both require proving that an execution with a particular dependency graph is in SI, rather than the other way round. We also hope that our specification of SI will be beneficial in other domains where dependency graphs have been useful, such as run-time monitoring [9, 37] and proving the correctness of concurrency-control algorithms [16, 36]. Finally, we expect that the approach to constructing a total commit order from transactional dependencies in the proof of our soundness theorem can be used to give dependency graph characterisations to other consistency models whose formulation includes similar total orders, such as prefix consistency [34].

The constraint on dependency graphs that we use to characterise SI also arose in the work of Lin et al. [24], who used it to formulate conditions under which a replicated database guarantees SI provided every one of its replicas does so. In comparison to them, we solve a more general problem of characterising SI regardless of how it is implemented and handle a variant of SI that does not require transactions to see the latest snapshot.
Transaction chopping has recently received a lot of attention. In particular, researchers have demonstrated that transactions arising in web applications can be chopped in a way that drastically improves their performance when executed under serializability [26, 36, 38]. There have also been proposals of consistency models for transactional memory that weaken consistency guarantees in a way similar to chopping [4, 20, 35]. Our chopping analysis enables bringing these benefits to transactional systems providing SI. We have previously proposed a chopping analysis for parallel SI [12], which also relies on a dependency graph characterisation of this consistency model (Theorem 21). But since parallel SI can be formulated without using an analogue of SI’s commit order, its dependency graph characterisation did not present the challenges that we had to deal with when establishing our soundness theorem.

Acknowledgements. We thank Hongseok Yang, who participated in early stages of this work. We also thank Hagit Attiya, Ricardo Jiménez-Peris and Pierre Sutra for comments that helped improve the paper. This work was supported by an EU project ADVENT.

References